Webservices in Morfik

Michaël Van Canneyt

October 9, 2007

Abstract

In the previous 2 articles about getting started with Morfik, the basics of writing a GUI interface and database access in Morfik were discussed. In this third part, the third pillar of Morfik will be discussed: building and using Webservices - also called webmethods in Morfik.

1 Introduction

Often, it is not enough to display forms with the result of queries: quite often, it is necessary to let the server execute some small task and have it return a result, without having to display a new webpage. This is an important part of any modern web application, and the cornerstone of AJAX technology. Morfik web applications are no exception to this rule. Except that in Morfik, the gory details of the sending of a webservice request are nicely wrapped in the Morfik framework. It's not necessary to know the JavaScript mechanisms involved, or the XML and SOAP message passing that is usually involved in the process.

In this article, the Morfik way of coding an Webservice (or Webmethod, as it is called in Morfik) will be examined. It will also be explained how to call external webservices from the browser and from the morfik server application.

2 Creating a webmethod

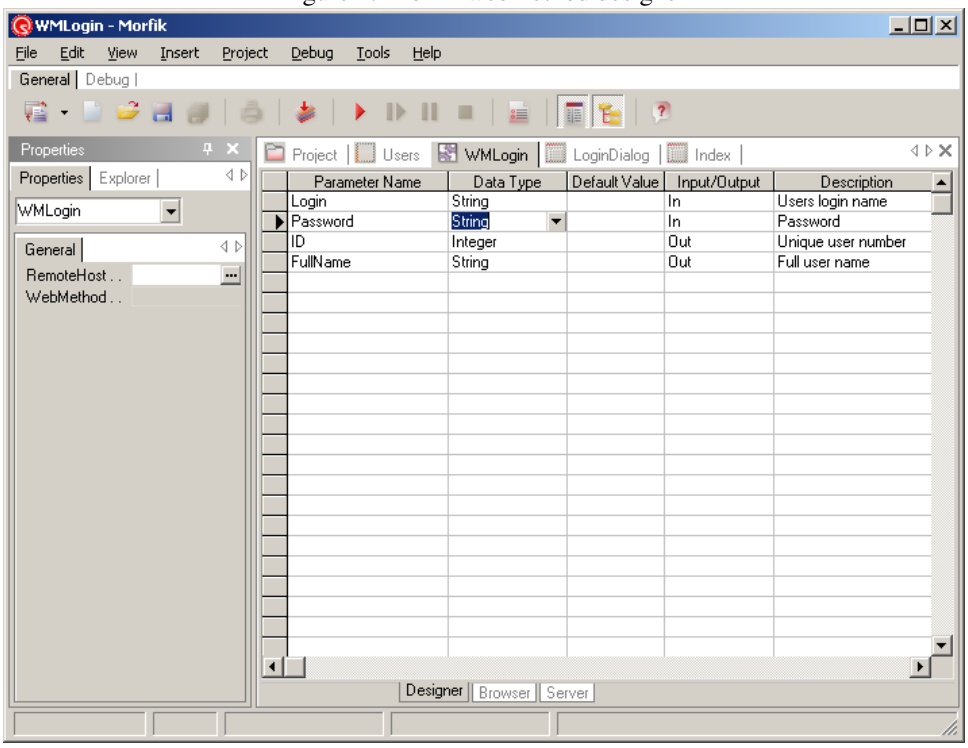
To demonstrate how to call a webmethod on the Morfik server application, a simple login routine will be coded. This is simple to do, and nevertheless demonstrates nicely the concepts involved.

The webservice demo program contains a simple form, with some labels on it. The `LLogin` label will be used to start the login or logout sequence. A second label `LUser` shows the currently logged in - none initially. When the `LLogin` label is clicked, the following code is executed:

```
Procedure Index.LLoginClick(Event: TDOMEvent);
Begin
  If (UserID=-1) then
    ShowLoginDialog
  else
    DoLogout(True);
End;
```

The `UserID` is the ID of the currently logged in user. If it is -1 (its initial value) then no-one is logged in, and the login dialog should be shown. If it has another value, then

Figure 1: Morfik webmethod designer



someone was logged in, and should be logged out. The login is performed using a popup dialog:

```
Procedure Index.ShowLoginDialog;

begin
  OpenForm('LoginDialog','popup','"modal=true","title=Log in"');
end;
```

The `LoginDialog` is a simple form: it contains 2 edits and 2 buttons, plus some labels. The edits serve to enter the username (`EName`) and password (`EPassword`). One button simply cancels the dialog, the other should perform the login. An invisible label is present (`LAuthenticating`) which will be made visible while the authentication is running, to provide some visual feedback to the user that authentication is in progress.

After this was created and configured, the login webmethod (named `WMLogin`) can be created.

As with all objects in Morfik, a webmethod can be created using a wizard, or directly in the designer. The designer serves to specify the in and output parameters to the webmethod, and is shown in figure 1 on page 2. As can be seen from the figure, the editor can be used to specify input and output parameters of the webmethod, together with their types, and a default value. Descriptive text can also be added. The editor currently supports only simple types: records, arrays are not yet supported, only the basic types such as integer, boolean, string, double and currency types are supported. Based on the parameters defined in the designer, the morfik IDE creates a class with the name of the webmethod: it creates both a server class and client (browser) class: all parameters defined in the designer are defined as fields in both these classes. The bottom of the figure shows the tabs `Server` and `Browser` in which these classes are defined.

In the server class, the `Execute` method should be implemented: this method should perform the action which the webmethod is supposed to do. For the login procedure, 4 parameters have been defined:

Login an input parameter with the login name of the user.

Password an input parameter with the password of the user.

ID an output parameter. On return, it will be filled with a unique numerical ID identifying the user, or -1 if the user is not known.

FullName an output parameter. On return, it will be filled with the full name of the user.

To code the login webmethod, a simple table 'Users' is created with 4 fields, corresponding to these parameters. The `ID` field is a simple autoincremental number, and the other fields are strings. Some sample data should be entered in the table.

Similarly, a query is created which will perform a lookup on this table:

```
SELECT
    ID, "FullName"
FROM
    "Users"
WHERE
    ("Login" = :Login) AND ("Password" = :Password)
```

The Morfik IDE will recognize the `:Login` and `:Password` as parameter values for the query. The query is saved as `QLogin`, and can now be used to code the `Execute` procedure of the webmethod.

The morfik IDE has created an empty `Execute` procedure in the server part of the webmethod, all that needs to be done is fill it with some code:

```
Procedure WMLogin.Execute;

Var
  Recordset : TRecordSet;
  DB : TIBOServiceConnection;

Begin
  ID:=-1;
  FullName:='';
  DB:=SoapServer.DefaultDBConnection;
  RecordSet := DB.CreateRecordSet('QLogin');
  Try
    RecordSet.Prepare;
    RecordSet.ParamByName('Login').AsString:=Login;
    RecordSet.ParamByName('Password').AsString:=Password;
    RecordSet.Active:=True;
    If Not RecordSet.EOF then
      begin
      ID:=RecordSet.FieldByName('ID').AsInteger;
      FullName:=RecordSet.FieldByName('FullName').AsString;
      end;
    RecordSet.Active := False;
    Finally
```

```
    DB.DestroyRecordSet(RecordSet);
  end;
End;
```

The code starts out by initializing the return values. After this, the connection to the default database is retrieved from the global server object `SoapServer`: this is a class which handles SOAP requests (a webmethod is always a SOAP request). The `CreateRecordSet` method of the database connector returns a `TRecordSet` class: this class is similar to the `TDataset` class in Delphi, and represents a cursor on the data. The `'QLogin'` value for the parameter tells the connection that a cursor for the pre-defined `QLogin` query should be returned. The parameters for the query are filled in and the query is executed. If it returns a result set (`EOF` determines wether the end of the result set was reached), then the fields are read from the cursor, and stored in the webservices parameters.

The browser part of the webmethod is more tricky to code. Webmethods are asynchronous in the browser: this means that the client code continues to execute while the WebMethod is being executed. The browser code is notified when the server returns a result. It is then up to the browser code to handle the result of the webmethod. The result of the webmethod should be handled in the `HandleResponse` part of the browser class.

To understand this better, let's have a look at how the LoginDialog calls the webmethod, in the `OnClick` handler of its `BLogin` button:

```
Procedure LoginDialog.BLoginClick(Event: TDOMEvent);

Var
  P : String;

Begin
  P:='"Login='+EName.Text+'",';
  P:=P+'"Password='+EPassword.Text+'"';
  Xapp.RunWebMethod('WMLogin',P,1);
  LAuthenticating.Visible:=True;
End;
```

The `RunWebMethod` of the client-side global `XApp` class will run a webmethod. It is declared as follows:

```
Procedure RunWebMethod(MethodName : String;
                       Params : String;
                       Encrypt : Integer);
```

The first parameter is the name of the webmethod to call. The second parameter should contain the parameters to the webmethod, specified as a comma-separated list of `Name=Value` pairs, enclosed in double quotes. The third parameter specifies whether the parameters should be encrypted - useful for instance when transmitting passwords, as in the above case.

Coming back to the login code above, the meaning of the code should now be clear. The observant reader will have seen that the `Visible` property of the `LAuthenticating` label is set only *after* the call to `RunWebMethod`. This is because the `RunWebMethod` is asynchronous: it returns immediatly after the request was sent to the server, without waiting for a response.

Internally, the `RunWebMethod` has created a `WebMethod` descendent instance (in the case of the login webmethod, a WMLogin instance), which handles the webmethod call. When the server has sent it's response back, the `WebMethod` class will parse the response

and store any output parameters of the webmethod in the appropriate fields. After this, the `HandleResponse` method will be called to handle the results of the webmethod. For the login webmethod, the code looks as follows:

```
Procedure WMLogin.HandleResponse;

Var
  L : TWebForm;

Begin
  L:=Xapp.Forms['LoginDialog'];
  If (ID=-1) then
    begin
    ShowMessage('Your username and password are not known!');
    LoginDialog(L).LAuthenticating.Visible:=False;
    end
  else
    begin
    L.Close;
    L:=Xapp.Forms['Index'];
    Index(L).UserID:=ID;
    Index(L).UserName:=FullName;
    Index(L).DoLogin;
    end;
End;
```

First, the `LoginDialog` instance is retrieved: the `XApp` class maintains a list of forms in the `Forms` array, which allows to access the forms by their name. If the login was unsuccesful, a message is shown, and the `LAuthenticating` label is made invisible. As a result, the user is simply left with the login dialog.

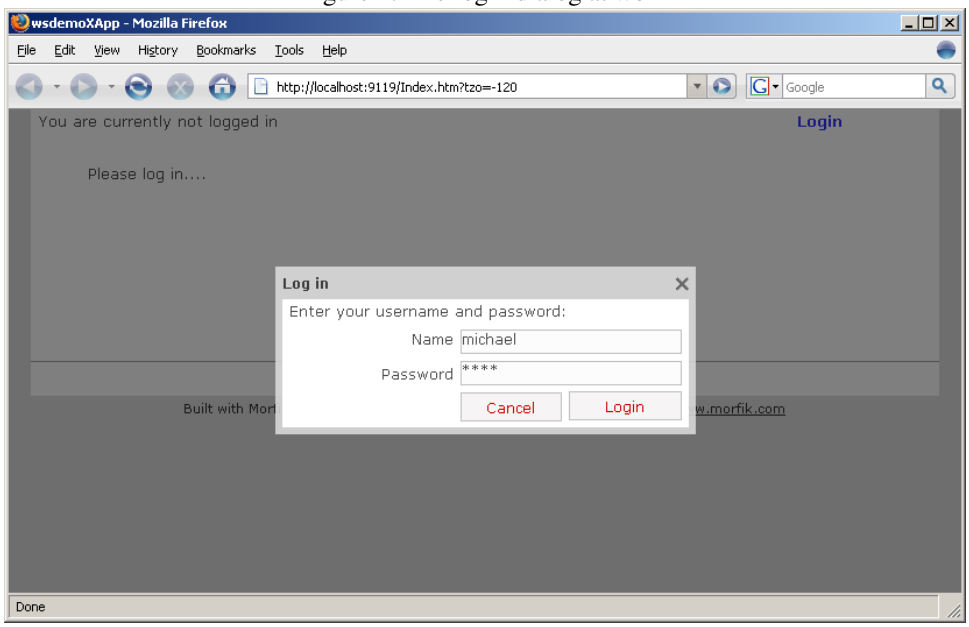
If the login was succesful, then the login dialog is closed, and the main form is notified that the user logged in: the `UserID` and `UserName` fields are filled in, and the `Login` method is called.

Whatever the result, after the `HandleResponse` method returns, the `WMLogin` instance is simply destroyed.

Due to the asynchronous nature of the webmethod, it is perfectly possible for the user to click on the 'Login' button again while the method is still running. This would set off a second request to the server. To avoid this, the `OnClick` handler can be changed:

```
Procedure LoginDialog.BLoginClick(Event: TDOMEvent);

Var
  P : String;

Begin
  If LAuthenticating.Visible then
    ShowMessage('Authentication in progress. Please wait...')
  else
    begin
    P:='"Login='+EName.Text+'",';
    P:=P+'"Password='+EPassword.Text+'"';
    Xapp.RunWebMethod('WMLogin',P,1);
    LAuthenticating.Visible:=True;
```

Figure 2: The login dialog at work




```
    end;
End;
```

If the webmethod call - for some reason - goes wrong, then the `LAuthenticating` label will of course remain visible, and the login dialog must be cancelled.

With this, all logic is ready - some setting of labels in the main form aside. The login dialog at work should look like in figure 2 on page 6.

3 Calling other webservices

Obviously, publicly available webservices should also be usable in a web application. Morfik allows the developer to do this: It is possible to import a webservice. To do so, a `WSDL` (Web Service Description Language) file is needed. if such a file is available, then it can be imported into Morfik using the **'Project - Import web services'** menu. This presents a small dialog that asks for a wsdl file. Once a file was selected, Morfik will create a module that contains a pascal declaration of the webservice.

This module is usable on the server only: the client cannot access webservices which are not located on the same server as the webpage it is viewing. This is not a Morfik restriction, but a JavaScript security restriction.

To access the third-party webservice on the browser, a webmethod must be created in the Morfik application that calls - from the server - the imported webservice. The browser will then call the webmethod, this in turn will call the actual third-party webservice, and will then transmit the results back to the client.

To demonstrate this, the google search webservice will be imported, and a small search facility will be added to the webservice demo. The google search WSDL file can be downloaded from Google. When using the Morfik webservices importer, a `googlesearchservice` unit is made which contains a definition for the exposed service:

```
TGoogleSearchServiceSoapIntf = Class(TSoapHttpClient)
```

```
  Function doGetCachedPage (key : string; url : string): String;
  Function doSpellingSuggestion(key : string; phrase : string): string;
  Function doGoogleSearch (key: string; q : string;
                           start: longint; maxResults: longint;
                           filter: boolean; restrict : string;
                           safeSearch : boolean; lr : string;
                           ie : string; oe : string):
                          TGoogleSearchResult;
End;
```

This class can be used to code the webmethod server part. We'll name the webmethod `WMGoogleSearch`, and give it an imput parameter `Query` and output parameter `QueryResult`, both of type `String`.

The `DoGoogleSearch` method of the Google Search Webservice is the method to use:

```
uses systemxml,googlesearchservice;

Procedure WMGoogleSearch.Execute;

Const
  Google_Web_Service_Key = 'xyz';

Var
  E  : TResultElement;
  S,ES : String;
  i  : Integer;
  Intf : TGoogleSearchServiceSoapIntf;
  Res  : TGoogleSearchResult;

begin
  intf:=TGoogleSearchServiceSoapIntf.Create;
  Try
    res := Intf.doGoogleSearch(Google_Web_Service_Key, Query,
                               0, 10, False, '', false,
                               '', '', '');
    If res.resultElements.item.Count = 0 Then
      S:=Enquote(XmlString(res))
    else
      Begin
      S:=IntToStr(res.estimatedTotalResultsCount);
      S:='Found '+S+' results:';
      For i := 0 To res.resultElements.item.Count - 1 Do
        Begin
        E:=TResultElement(res.resultElements.item[i]);
        ES:='<b>URL</b> : <u>'+E.URL+'</u><br>' +
            '<b>title</b> : '+E.Title + '<br>' +
            '<b>snippet</b> : '+E.snippet;
        S:=S+'<br>'+ ES;
        End;
      End;
    QueryResult:=S;
  Except
    QueryResult:=Exception(ExceptObject).Message
  End;
```

```
  Intf.Free;
End;
```

The code is rather straightforward: an instance of the google search interface is created, and the `doGoogleSearch` call is executed. The first parameter is a key which must be issued by Google (for the purpose of this article, the Morfik key was used). The second parameter is the search term, and the next 2 parameters specify that the first 10 search results should be returned. All other parameters are not relevant to the discussion here.

The `doGoogleSearch` actually does the complete webservice call to google. After it returns, the search result is in `res`. The rest of the webmethod `Execute` call is just formatting a nice result string, which can be shown in the browser. The code is pretty straightforward, even without knowing the details of all classes involved in the result description. The last step is just storing the result string in `QueryResult`.

The browser code for the `WMGoogleSearch` webmethod is very simple:

```
Procedure WMGoogleSearch.HandleResponse;
Begin
  Index(XApp.Forms['Index']).HandleSearchResult(QueryResult);
End;
```

The `HandleSearchResult` of the `Index` form will simply apply the result to a label:

```
Procedure Index.HandleSearchResult(AResult : String);

begin
  LResult.Caption:='<PRE>'+AResult+'</PRE>';
end;
```

To call the search webmethod, an additional edit control (`EQuery`) and a button (`BSearch`) are placed on the form. The `OnClick` handler of the form gets the following code:

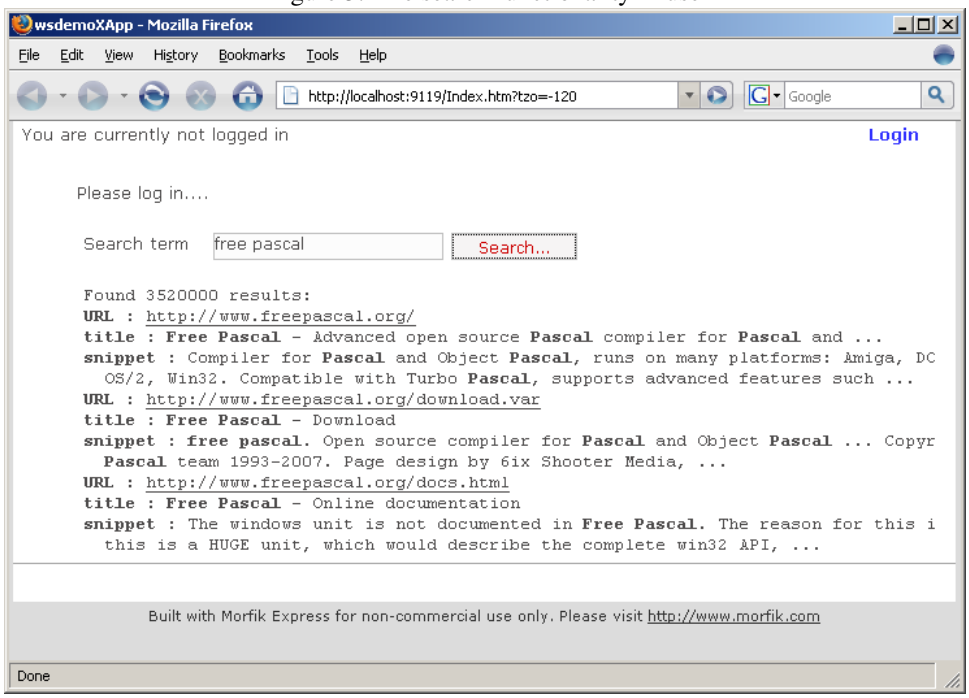
```
Procedure Index.BSearchClick(Event: TDOMEvent);

Var
  P : String;

Begin
  P:='"Query='+EQuery.Text+'"';
  Xapp.RunWebMethod('WMGoogleSearch',P,0);
End;
```

Which introduces nothing really new. This completes the implementation of accessing the Google search API. The result can be seen in figure 3 on page 9.

4 Conclusion

This article was meant to show that the use of webservices in Morfik is very easy, both if implementing it in Morfik or when calling third-party webservices. The implementation as it is presented here has a few drawbacks, though: the first is the asynchronous nature of the call, which makes programming a webservice rather awkward for a desktop application programmer. Secondly, the call to a webservice or webmethod is rather cumbersome: the encoding of the parameters is rather strange, and does not allow for type checking. Accessing a third-party webservice from the browser is currently also not yet as straightforward as

Figure 3: The search functionality in use



it could be: The webmethods currently allow only simple types to be passed, requiring an additional webmethod per accessed webservice. The Morfik developers are aware of this and have remedied all these issues: An upcoming release (not yet available for testing at the time of writing of this article) will address all these issues and many more, making the use of webservices even more simple than what has been shown here.